1) Data types & its functions:

Numeric → int, float, complex
Text → string
Sequence → list, tuple (also string)
Mapping → dictionary
set → set, frozen set
Boolean → bool

* Mutable → list, dictionary, set
* Immutable → int, float, decimal, bool, string, Tuple

2) Types of Arguments:

- Positional
- Default
- Keyword
- Arbitrary

3) Python Functions
4) Recursive function
5) Exceptional Handling in Python
6) File Handling
7) Regular Expressions
8) Lambda, Maps, Filters, Reduce
9) Difference between:
   1) Shallow copy & deep copy
   2) range() and xrange()
   3) *arg and **kwargs
   4) Pickling and Unpickling
10) Python Memory Management
    - Garbage collector

11) Iterators and generators

12) Decorators

13) Python Closure.

14) Various Python Modules:
- → os module
- → re module
- → subprocess module.
- → logging module
- → date and time → Imp

15) OOPs

16) classes and Objects

17) Constructors and destructors

18) self keyword

19) Inheritance and its types

20) Encapsulation

21) Polymorphism
- Overloading & overriding

22) super() keyword

23) Method Resolution Order (MRO)

24) Python Multithreading & Multiprocessing.

25) Python Collections
- Ordered dict
- Chainmap
- named tuples
- deque

Python

→ High level language
→ Both programming and scripting language.
→ Free and open source
→ Easy & User friendly, almost like English language
→ Strong OOPs concepts
→ Wide variety of variables, modules & libraries.

Variables

→ Rules:-

- starts with letter or underscore
- Do not use reserve words (keywords)

→ Multiple assignment

```
var1 = var2 = var3 = 500

a, b, c = 100, 200, 300
```

3) String " " or ' '

Sequence of character data type.

Indexing & Slicing

Eg:-

```
str = "Hello"
print(str[0])
```

output ⇒ H

```
print(str[3])
```

output ⇒ l

index()
count()
len()
upper()
lower()
isupper()
islower()
swapcase()
replace()
enumerate()
strip()
split()
join()
Concatenation & Membership

enumerate()

→ returns an enumerate object.

It contains the index and value of all the items of the string as pairs.

len() → returns the length of the string.

Eg-

```
str = 'cold'
list_enumerate = list(enumerate(str))
print('list =', list_enumerate)
print('length =', len(str))
```

→ output

list = [(0,'c'), (1,'o'), (2,'l'), (3,'d')]

length = 4

index() ⟷ find()

→ gives the index of the specified character

eg:- str = "Hello"

print (str.index("e"))

→ output ⇒ 1

str.index("o") ⇒ 4

count()

→ gives the no. of times the specified character is present in the string.

eg:- str.count('l') ⇒ 2

str.count('H') ⇒ 1

upper() → converts ~~all~~ the entire string into uppercase

lower() → converts entire string into lowercase

isupper() → gives Boolean - True/False

True → if the string is in Upper

islower()

swapcase()

replace('m','n') → replaces the m^{th} character by the specified 'n' character.

```
str = "Hello"
str2 = str.replace(str[2], "z")

output (str2)
```

⇒ Hezlo

strip() → strips the whitespaces at each ends of the string.
Also, removes specified characters from ends of the string.

Eg:-
```
str = "  Python  "
print (str.strip())
```
⇒ output ⇒ "Python"

```
str = "Python"
print (str.strip ("Pyon")
```
⇒ th

split() → splits the string at the specified character & puts it into a list.
Splits at 'whitespace' if no character is specified.

Eg:- ~~str = "Python"~~ str = "Py,thon"
```
print (str.split(','))
```
⇒ output ⇒ ['Py', 'thon']

join() → returns a string concatenated with the elements of an iterable.

Eg:-
```
str1 = "Hello"
str2 = "123"
new_string = str1.join(str2)
print (new_string)
```
⇒ output ⇒ 1Hello2Hello3Hello

String Concatenation :- (+)

- Adds 2 strings

* Repeat string (*)

Repeats a string for specified no. of times

Eg:- str1 = "Hi"
str2 = "World"
print ("concatenation=", str1 + str2)
print ("Repeatation=", str1 * 3)

⇒ output

HiWorld
HiHiHi

String Membership : (in and not-in)

>>> 'a' in 'program'
True
'at' not in 'battle'
False

String Formatting: { }

str1 = "Hello"
str2 = "World"
print ("New String = {0}, {1}".format(str1, str2))

⇒ output ⇒ New String = Hello, World

* Positional Arguments
* Keyword Arguments

String Slicing :-

```
          0 1 2 3 4 5 6 7 8 9 10
str = "Prem Kumar"
      -10 -9 -8 -7 -6 -5 -4 -3 -2 -1
```

print (str [1:9]) ⇒ "rem Kuma"

print (str [2:5]) ⇒ "em"

[0:9] ⟷ [:9] ⇒ "Prem Kuma"

Negative Slicing :-

print (str [-3:]) ⇒ mar

print (str [-10:-5]) ⇒ "Prem"

Reverse a String :-

print(str[::-1]) ⇒ ramuk merP

print (str [3:0:-1]) ⇒ mer

List [ ]

List is a changeable sequence of data elements.

* Indexing and Slicing

```
my_list = [1, 2, 3, 4]
print (my_list[0])  ⟹ 1
print (my_list[3])  ⟹ 4
print (my_list[-3]] ⟹ 2

my_list = [1, 2, 3, [4, 5, 6])
print (my_list[3][2]) ⟹ 6
```

Slicing :-

```
print (my_list[1:3] ⟹ [2, 3]
```

List Concatenation & Repetation

```
list1 = [1, 2, 3, 4]
list2 = ["H", "i"]

print (list1 + list2)
print (list2 * 3)
```

output ⇒ [1, 2, 3, 4, 'H', 'i']

output ⇒ ['H', 'i', 'H', 'i', 'H', 'i']

List Operations

Indexing

Slicing

index()

len() → no. of times the character is present.

count()

append() → add an element at the end of list

extend() → used to add another list into this list

insert() → add element at specified index

del → delete entry from the list

pop() → removes the element at specified Index

remove() → removes the given Value

clear() → clears all the ele from the list

sort() → sorts the list in ascending order

reverse() → reverses the string

copy() → returns a shallow copy of the list.

max() → returns the max element of the list

min() → min element in the list

li.pop(index) ⟷ del (li[index])

list1 = [1, 2, 3]

print(len(list1)) = 3

list1.append(4) ⇒ [1, 2, 3, 4]

list1.insert(2, "Hi")

⇒ [1, 2, "Hi", 3]

Eg:- list1 = [1, 2, 3, 4]

list1 [1:5] = [7, 8]

⇒ [1, 2, 3, 4, 7, 8]

list1 = [1, 2, 3, 4]

list1.extend([9, 10, 11])

~~⇒ [1, 2, 3, 4, [9, 10, 11]]~~

output ⇒ [1, 2, 3, 4, 9, 10, 11]

list1 = [1, 2, 3, 4]

list1.pop(1)

print(list1)

⇒ [1, 3, 4]

list2 = [4, 5, 6]

print(list2.pop()) ⇒ [4, 5]

list 1 = [1,2,3,4]

list1.remove(2)

print(list1) ⇒ [1,3,4]

list 1 = [1,2,3,4]

del list1[1] ⇒ [1,3,4]

del list1 = ~~[]~~ deletes the entire list

Tuple ( )

It is similar to list

It is Immutable.

* Difference b/w list and Tuple :-

We cannot change the elements of a tuple once it is assigned, whereas in 'list' elements can be changed.

Count and Index Only & Slicing

Tuple Membership.
Tuple Concatenation
Tuple Repetition

index (x) → Returns the index position of the element x
count (x) → Returns the no. of times the element 'x' is present in the tuple

len()
cmp()

tuple_name = tuple(list_name) → converts list into tuple

To assign single value into tuple ⇒ a = 1, or a = (1,)

* catch-all variable
_ disposable variable

Packing and Unpacking.

Packing :-

```
my_tuple = 3, 4, 5.6, "Hello"

print(my_tuple)
        ⇒ 3, 4, 5.6, "Hello"
```

Unpacking :-

```
a, b, c, d = my_tuple
print(a)  ⟶ 3
print(b)  ⟶ 4
print(c)  ⟶ 5.6
print(d)  ⟶ "Hello"
```

Catch-all variable

```
a, *more, last = my_tuple
a = 3
more = [4, 5.6]
last = "Hello"
```

```
_, b, c, _ = my_tuple
```

_ ⇒ disposable variable. (like if you need only some elem of a tuple)

Advantages of Tuple :-

→ Heterogeneous

→ Since it is immutable, iterating over the tuple is easy & hence a performance boost

→ Since immutable, can be used as a key for dictionary.

→ Since immutable → Write Protected.

---

example for tuple packing & unpacking

```
my_tuple = ("GSSS", 100, "CS")
(college, students, branch) = my_tuple

print(college)  ⇒ GSSS
print(students) ⇒ 100
print(branch)   ⇒ CS
```

Dictionary {key:value}

Dictionary is unordered collection of items

- Not Sequenced
- Key:value relationship
- Key should be Unique and cannot be a lis
- Value can be anything.

Since it is not Sequenced, Indexing is not possible

To add element into a dictionary:-

```
my_dict = {'name':'Jack','age'=26}
my_dict['address'] = 'US'
print(my_dict) ⟹ {name=Jack, age=26, address=US}
```

→ `dict_name[key] = value`

sorted(), all(), any(), cmp()
len()
get() ⇔ indexing.

keys() → Returns a new view of dict keys
values() → Returns a new view of dict values
items() → Returns a new view of dict items (key,valu

→ pop(key) → Removes the particular key from dict & returns the value as result
popitem() → Removes arbitrary (key,value)
→ del dict_name[5]
clear()

Dictionary Comprehension:

Creates a new dictionary from an iterable

Eg:-

```
squares = {x : x*x for x in range(6)}
print (squares)
```

⇒output⇒ {0:0, 1:1, 2:4, 3:9, 4:16, 5:25}

Dictionary Membership :-

Returns Boolen True or False

```
squares = {0:0, 1:1, 2:4, 3:9, 4:16}
print (2 in squares)
True
print (3 not in squares)
False.
```

---

len()
sorted()
all()
any()
cmp()

get()
keys()
values()
items()
pop(key)
popitem()
clear()
del()

Set { }

- Set is Unordered collection of items
- Every element is Unique

→ Elements are Immutable (cannot be changed)

However, Set itself is mutable
We can add or remove items from it.

While creating an empty set, declare it as Set

Eg:- a = set()

add() → add single element

update() → adds multiple elements

discard(x) → deletes an element 'x' from set.
If element is not present in set, set remains unchanged.

remove(x) → If element is not present, throws Error.

pop() → Remove and return that arbitrary element

~~Method~~

```
my_set = {1, 3}
~~print~~
my_set.add(2)
print(my_set)  ⟶  {1, 2, 3}

my_set = {1, 2, 3}
my_set.update([4,5])
print(my_set)  ⟶  {1, 2, 3, 4, 5}
```

Set Operations

Union

Intersection

Difference

Symmetric Difference

Set Union |

Union of A & B is all the elements from both the sets

A = {1, 2, 3}

B = {3, 4, 5, 6}

print (A|B) or print (A.union(B))

output ⇒ {1, 2, 3, 4, 5, 6}

Intersection : &

. Elements common in both A & B.

A = {1, 2, 3, 4}

B = {3, 4, 5, 6}

print (A&B) or print (A.intersection(B))

⇓

{3, 4}

Difference :- −

A − B ⇒ elements in A but not in B

B − A ⇒ elements in B but not in A

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}

print (A − B) or print (A.difference(B))

⟹ {1, 2, 3}

print (B − A) ⟹ {6, 7, 8}

Symmetric difference ^

Symmetric difference of A and B is a set of elements in both A and B except those are in common.

A = {1, 2, 3, H}

B = {3, H, 5, 6, 7}

print (A ^ B) or print (A.symmetric_difference(B))

⇒ output ⇒ {1, 2, 5, 6, 7}

Set Membership

my_set = set("apple")

print ('a' in my_set) ⇒ True

print ('p' not in my_set) ⇒ False.

Frozen Set

Frozen set is a new ~~characte~~ class that has the characteristics of a set, but its elements cannot be changed once assigned.

Frozensets are immutable Sets.

Sets being mutable are unhashable. So they can't be used as dictionary keys.
On the other hand, frozensets are immutable are hashable and can be used as keys of the dictionary.

Frozenset can be created using the function

frozenset()

Methods:-

copy()
union()
intersection()
difference()
symmetric_difference()
issubset()
issuperset()
isdisjoint()

Function Arguments

1) Position Arguments
2) Default Arguments
3) Keyword Arguments
4) Arbitrary Arguments

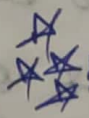
```
def greet(name, msg):
    print("Hello", name + ',' + msg)

greet("John", "Good Morning")
```

output ⇒ Hello John, GM

① Positional Arguments

name = John
msg = GM

② default Arguments



Once we have the def arg, all the args to its right must be def args.

```
def greet(name, msg="GM")
    print("Hello", name + ',' + msg)

greet("Jon")
greet("Kev", "Hey")
```

default →

If nothing is specified while calling a function, then it takes the default arg.

If any other value is provided, then it will overwrite the default value.

⑤. Keyword Arguments

When we call a function, it takes the positional arguments

In python, we can change the order of the arguments while calling a function.

Eg:- greet (name = "John", msg = "GM")
greet (msg = "Hey", name = "Keer")

** Keyword Arg must follow Positional Arguments.

⑥. Arbitrary Arguments.

When we don't know the no. of arguments to be passed to a function, we use * symbol for the argument.

Eg:- def greet (*names):
    for name in names:
        print ("Hello:", name)

greet ["Keer", "John", "Monica"]

Recursive Function :-

A function calling itself.

Eg:- Factorial of a number.

```
def factorial_num(x):
    if x==1:
        return x
    else
        return (x= x * factorial_num(x-1))

num=4
factorial_num
print("Factorial of {} is {}".format
                    (num, factorial_num(num))
```

Exception Handling :-

Python does not have compilation.
Everything is on runtime only.

An Exception is an error that happens during the execution of the program.

When that error occurs, python generates an exception that can be handled, which avoids your program to crash.

When you think you have a code, which can produce error then you can use exception handling.

["try and except clause"]

Syntax:

```
try:
    <code to be executed>
except:
    <raise exception if there is errore>
else:
    <execute this code if there is no error>
finally:
    <execute this no-matter-what also do some cleanup>
```

↓

(close UI, db, file---)

```
try:
    :
    :
except Exception as err:
```

↓

Superclass for all the exceptions.

If we don't specify the particular type of exception, it will catch all the exceptions, which is a bad idea because the program will ignore the unexpected errors along with the ones the 'except' block is actually prepared to handle.

You can also raise an Exception using the keyword "raise"

Eg:-
```
x = -1
if x < 0:
    raise Exception("Sorry, no numbers below zero")
```

Types of Exceptions:-

IOError → If file cannot be opened

ImportError → If python cannot find the module

ZeroDivisionError

IndexError

ValueError

TypeError

NameError

ArithmeticError

KeyError

Exceptional Handling

Eg:-

```
def trial (a,b):
    try:
        c = ((a+b)/(a-b))
    except ZeroDivisionError:
        print ("Can't divide by zero")
    else:
        print c
    finally:
        pass
```

trial (3,3) ⟹ will raise the ZDE exception

trial (4,2) ⟹ will print 'c' (else part)

File Handling

File is a named location on disk to store related information.

File operation takes place in the following order

- Open a file
- Read or write (perform operation)
- Close the file.

Syntax

with open ('filename', 'accessmode') as somename

Accessmodes:

r ⇒ open a file for reading.
w → writing
a → appending at the end of the file without truncating it

t → open in text mode (default)

b → open in binary mode

'+' → open a file for updating.

***

NOTE If no access_mode is specified, by default the file will take read-mode (r)

File Operations:

write() → to write into a file. Writes string 's' to the file & returns the no. of char written

read() → read till end of file

read(size) → read the specified size of characters from the file.

seek() → change the current cursor position

seek(0) → bring cursor to initial position.

tell() → returns the current cursor position.

readline() → read individual lines of a file. This method reads a file till the newline, including the newline character.

readlines() → returns a list of all remaining lines of the entire file.

writelines(lines) → write a list of lines to the file

fileno() → returns an integer no. of the file.

close() → closes a file

Regular Expressions

Import the module re

Sequence of characters that matches a string.

[Metacharacter → character intepreted in a special way]

^ → starts with

$ → ends with

. → one occurance of a character

* → zero or more occurance

[ ] → set of characters matching (atleast one inside the brackets)

[a-e] - → range ⇒ [abcde]

[0-9] ⇒ [0123456789]

[^abc] → any characters except a, b or c

+ → one or more occurances of pattern left to it

eg: ma+n

match: man, maaan

? → zero or one occurance of pattern left to it

eg: ma?n

match ⇒ mn, man

no match ⇒ maaan

{m,n} → atleast m and atmost n times repetition of the pattern

( ) → used for grouping

| → 'OR' operation

\ → escape the meaning of character.

\A → matches if specified characters are at the begining of a string

\Z → matches if specified char are at end of the string

\b → matches if specified char ARE at beg or end of a WORD

\B → — " — are-not at beg or end of a word

\d → [0-9]

\D → non-decimal digit

\s → white space character in a string

\S → non-white characters in a string

\w → alphanumeric

\W → non-alpha numeric

Regular Expression Methods:

findall()

split() → maxsplit

sub()

subn()

search()

match() → group()

match.group()
match.start()
match.end()
span()

findall() → returns a list of string containing all matches

eg:
```
import re
string = "Hello 12 hi 89"
pattern = "\d+"
result = re.findall(pattern, string)
print(result)
```

⇒ output → ['12', '89']

split() → splits the string where there is a match and returns a list of strings where splits occurred.

eg:
```
string = "Hello:12 Eighty nine:89."
pattern = "\d+"
result = re.split(pattern, string)
print(result)
```

⇒ output ⇒ ['Hello:', ' Eighty nine:', '.']

maxsplit → no. of splits that will occur.

result = re.split(pattern, string, 1)

output = ['Hello:', 'Eighty nine:89.']

sub() → returns a string where match occurances are replaced with the content of replace variable.

Eg:- import re

string = "Hello World Hi"

pattern = '\s+'

replace = 9

result = re.sub(pattern, replace, string)

print(result)

output ⇒ Hello9World9Hi

subn() → similar to sub(), but it returns a tuple of 2 items containing the new string and the no. of substitutions made.

for the above example, if subn() is called then output will be:

('Hello9World9Hi', 2)

re.search()

Takes 2 arguments : a pattern and a string. Looks for the first location where RegEx pattern produces a match with the string.

If the search is successful, it returns a match object ; if not returns None.

Eg:-

```
impost re
string = "Python is fun"
match = re.search('\APython', string)
if match:
    print ("Pattern found")
else:
    print ("Not found")
```

re.search returns a match object only. We use group() or groups() to retrieve the matched expression

---

re.match()

same as search, but match() searches for the pattern only in the first line of the string or only at the beginning.

---

match.group()
match.start()
match.end()
match.span()
match.re
match.string

Lambda function :-

Anonymous function

Syntax :-

lambda arguments : expression

Eg:-

```
double = lambda x : x * 2
print (double (5))
```

(1) Filter ()

Takes function and a list as arguments. Function is called with all the items in the list & returns a new list with items for which the function evaluates True.

eg:-

```
list1 = [1, 2, 3, 4, 5, 6, 7, 8]
new_list = list (filter (lambda x : (x % 2 == 0), list1))
print (new_list)
```

⇒ output ⇒ [2, 4, 6, 8]

(2) Map()

Function is called with all the items in the list & returns a new list with all the items returned by that function for each item.

eg:- list1 = [1, 5, 2]

new_list = list(map(lambda x : x*2, list1))

print(new_list)

→ output ⇒ [2, 10, 4]

(3) reduce()

reduce() accepts a function and a sequence & returns a single value calculated as follows:

- Initially, the function is called with first 2 items from the sequence & the result is obtained
- The function is then called with the result obtained in step 1 & the next value in seq. & the process repeats

Syntax :- reduce(function, sequence[, initial])

```
from functools import reduce
def do_sum(x1, x2):
    return x1+x2
print(reduce(do_sum, [1,2,3,4,5]))
```

```
my_numbers = [4, 6, 9, 10]
result = reduce(lambda num1, num2 : num1 * num2, my_numbers)
print(result)
```

Shallow copy and deep copy:-

list 1 = [1, 2, [3, 4], 5]

new_list = copy.copy(list1) ⇒ shallow copy

new_list = copy.deepcopy(list1) ⇒ deep copy.

Deep Copy:-

Is a process in which copying process occurs recursively

It means, first constructing a new collection object and then recursively populating it with the copies of the child objects found in the original.

In case of deep copy, a copy of the object is copied in the other object.

It means that any changes made to a copy of obj, do-not reflect in the original object.

Shallow Copy:-

Constructing a new collection obj & then populating it with references to the child obj found in original.

Copying process won't recurse & hence won't create copies of the child obj themselves.

In case of shallow copy, reference of the object is copied in the other. Hence changes made to the copy of the object Do-reflect in the original.

8. range() and xrange()

range() → This returns a list of numbers created using range() function
It produces all the no, at once. Hence more memory.

xrange() → returns the generator object that can be used to display the numbers only by looping.
It produces the values/numbers only on demand & hence keeps only one value in memory at a time. Hence it consumes less memory & more faster.

Pickling and Unpickling

(Serialization / marshalling / Flattening)

Pickling is the process of converting a Python object into a byte stream and store it in a file/database, in order to maintain the program state across the sessions and re-use them at another ~~runtime~~ time without losing any instance specific data.

Unpickling is the reverse of pickling process where a byte stream is converted into an Object hierarchy.

Memory Management in Python

Python memory management involves a private heap space.

The allocation, de-allocation everything is managed internally by Python memory manager.

Python manages the object using reference counting.

This means that the memory manager keeps track of the number of reference count to each object in a program.

When the reference to an object reaches zero, which means the object is no longer used, the garbage collector automatically frees up the memory from that particular object.

The user never has to worry about memory management as everything is fully automatic.

Iterators and Generators

Iterators:-

Iterators are the objects that can be traversed or loop over.

In other words, you can run "for" loop over the object.

Examples for iterators: string, list, tuple

String is an iterator & you can run a for loop over

`__iter__` ⇒ returns the iterator object itself & is used while using "for" & "in" keywords.

`__next__` ⇒ returns the next value from the iterator.

Eg:-
```
string = "Hello"
my_iter = iter(string)
print(my_iterator) → shows the memory allocation of the iter
```

To use it, we have to use 'next' keyword explicitly

```
print(next(my_iterator))
```

It fetches only one value and awaits for next.

Generators:-

simple way of creating iterators.

Generator is a function that returns an object that we can iterate over.

Generator function consists of one or more "yeild" stat

'yeild' is used instead of 'return'.

Difference is that 'return' statement terminates a function entirely, where as the 'yeild' statement pauses the function saving all of its states and later continues from there on successive calls.

→ Generators

- It maintains its state so that the function can resume again exactly where it left.
- Produces a sequence of ~~resu~~ results.

Eg:-

```
def my_gen():
    n = 1
    print("This is printed first")
    yeild n
    n += 1
    print("This is printed second")
    yeild n
```

```
a = my_gen()
next(a)  ⇒ first
next(a)  ⇒ second.
```

Decorators

Also called as **Metaprogramming**.

Decorators are used to modify (change or the behaviour of a function or a class without permanently modifying that existing function.

In decorators, functions are takes as arguments into another function and then called inside the wrapping function.

Eg:-

```
def outer(func):
    def inner():
        print("I am decorated")
        func()
    return inner()

@outer
def ordinary():
    print("I am ordinary")

ordinary = outer(ordinary)
ordinary()
```

Python Closure:

Nested function:- A function defined inside another function.

The outer function is called 'Enclosing func

Nested functions can access the variables of the enclosing scope.

We can have a closure if it has following 3 condit

1) We must have a nested function
2) Nested function must access a value defined in the enclosing function.
3) The Enclosing function must return the nested functo

Eg:

```
def print_msg(msg):
    def printer():
        print(msg)
    return printer

another = print_msg("Hello")
another()
# print(another.__closure__)
```

→ Python closure:

The print_msg() function was called with a string "Hello". and the returned function was bound to the name another.
On calling another(), the msg : "Hello" was still remembered although we had already finished executing the print_msg() functio

This technique by which some data ("Hello") gets attached to the code is called closure

This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

All function objects have a __closure__ attribute that returns a tuple of cell objects if it is a closure function.

Python Date and Time

```
import datetime
```

Eg 1:-

```
import datetime
x = datetime.datetime.now()
print(x)
```

output ⇒ 2018-12-19 09:02:03.472603

Eg 2:-

```
import datetime
x = datetime.date.today()
print(x)
```

output ⇒ 2018-12-19

datetime.datetime.now()
↓ module ↓ class ↓ method

4 classes in datetime module:-

1) date class
2) time class
3) datetime class
4) timedelta class

date class

(1) from datetime import date

```
x = date(2020, 01, 13)
print(a)
```

⇒ 2020-01-13

(2) from datetime import date

```
x = date.today()

print (x.year)
print (x.month)
print (x.day)
```

Time Class :-

(1) from datetime import time

```
a = time()
print(a)
```

⇒ 11:34:56.234678

(2) from datetime import time

```
a = time (11, 34, 56)

print(a.hour)
print (a.minute)
print (a.second)
```

Datetime class

① from datetime import datetime

```
a = datetime (2018, 11, 28)
print (a)
b = datetime (2018, 11, 28, 23, 55, 59, 342380)
```

② from datetime import datetime

```
a = datetime()
print (a.year)
print (a.day)
print (a.minute)
print (a.second)
```

timedelta class

① from datetime import datetime, date

```
t1 = date (year = 2018 , month = 7, day = 12)
t2 = date (year = 2017, month = 12, day = 23)
t3 = t1 - t2
print (t3)

t4 = datetime (year = 2018, month = 7, day = 12, hour = 7,
               minute = 9, second = 33)

t5 = datetime (year = 2018 , month = 3, day = 02,
               hour = 5, minute = 10, second = 4

t6 = t4 - t5
print (t6)
print ("type of t3 = ", type(t3))
```

⇒ output ⇒

t3 = 201 days , 0:00:00

t6 = -33 days , 1:14:20

type of t3 = <class 'datetime.timedelta'>

---

strftime() → datetime object to string

```
from datetime import datetime
now = datetime.now()
t = now.strftime("%H:%M:%S")
print(t)
m = now.strftime("%m/%d/%Y, %H:%M:%S")
print(m)
```

output ⇒

t = 04:34:52

m = 12/26/2018, 04:34:52

%a → weekday in short ⇒ Wed

%A → weekday in full ⇒ Wednesday

%d → day no. ⇒ 01 to 31

%w → weekday no. ⇒ 00 to 06 (Sunday)

%b → month name in short ⇒ Dec

%B → month name in full ⇒ December

%m → month in number ⇒ 01 to 12

%y → year in short → 19

%Y → year in full ⇒ 2019

%H → Hours [00 to 23]

%M → Minutes [00 to 59]

%S → seconds [00 to 59]

strptime() ⟶ string to datetime

```
from datetime import datetime
date_string = "21 June 2018"
date_obj = datetime.strptime(date_string, "%d:%B:%Y")
print(date_obj)
```

Output ⟹ 2018:06:21

OOPs

DRY
Dont-Repeat-Yourself

The concept of OOP in Python focuses on creating Re-usable code

Classes and Objects :-

An object has 2 characteristics:
- Attributes
- Behaviours

For example; if Parrot is a class; then;
- name, age, color are attributes
- singing, flying are behaviours.

Class :-

A class is a blueprint for the object.

Object :-

An object is an instantiation of a class.
Object is nothing but collection of data & method

OOPs follows:-
- Inheritance
- Encapsulation
- Polymorphism.

Constructors, destructors :-

Constructors :-

Constructors are called when the object is created.
The task is to initialize the data members of the class when an obj of class is created.

__init__()

2 Types :-
1) Default Constructor → class A : def __init__(self):
2) Parameterized Constructor → def __init__(self, a, b, c):

Destructor :-

Destructors are called when an object gets destroyed.

__del__() method is known as destructor method.
It is called when the reference to obj are deleted
ie, when the reference count becomes zero.
In Python __del__() is not much needed because
Python has a garbage collector that handles the
memory management automatically.

```
class Employee:
    def __init__(self):
        print("Employee created")

    def __del__(self):
        print("Employee deleted: Destructor called")

obj = Employee()
del obj
```

Garbage Collection :-

Python automatically releases the memory
from the unused objects. It runs during
pgm execution & triggered when the obj
reference count becomes zero.

Self :-

Self represents the instance of the class.

We can access the attributes and methods of a class using the self keyword.

Whenever we call the method, the instance of the class (object) itself automatically passes as a first argument along with the other arguments of the meth

If no other arg are passed, only self is passed to the method.

Inheritance :-

It refers to defining a new class with little or NO modification to an existing class.

New class is called derived (child) class.

Old class is called base (parent) class.

Derived class inherits the feature from the base class, adding new features to it.

This results in re-usability of the code.

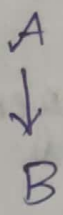
child classes Over-ride or extend the functionality of the parent class

Eg :-

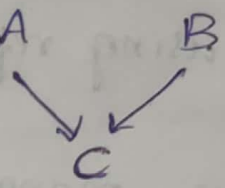
```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)

class Student(Person):
    pass

x = Person("Keer", "N")
x.printname()
x = Student("Sowmya", "Iyer")
x.printname()
```

Types of Inheritance

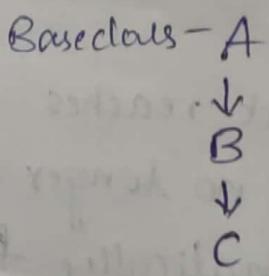
1) Single Inheritance



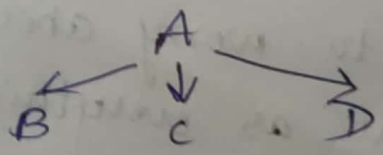
2) Multiple Inheritance



3) Multilevel Inheritance



4) Hierarchical Inheritance



5) Hybrid Inheritance

Combination of multiple types of inheritance.

Encapsulation :

Using OOP in Python, we can restrict the access to the methods and variables. This prevents the data from direct modifica

[Protected variables → _ / Private members → __]

Eg:-
```
class Computer:
    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print(self.__maxprice)

    def setPrice(self, price):
        self.__maxprice = price
```

```
c = Computer
c.sell()
c.__maxprice = 1000
c.sell()
c.setPrice(1500)
c.sell()
```

output :-
900
900 → value cant be changes as it is private
1500

Polymorphism :-

Same function name (but different signatures) being used for different types.

Example 1 :

```
class Birds:
    def intro(self):
        print("There are many birds")

    def flight(self):
        print("some fly, some cannot")

class Parrot(Bird):
    def flight(self):
        print("Parrots can fly")

class Penguin(Bird):
    def flight(self):
        print("Penguins cannot fly")
```

```
obj_bird = Bird()
obj_par = Parrot()
obj_pen = Penguin()

obj_par.intro()
obj_par.flight()
```

Polymorphism Example 2: —

```
class India:
    def capital(self):
        print("Delhi")

    def language(self):
        print("Hindi")


class USA:
    def capital(self):
        print("Washington DC")

    def language(self):
        print("English")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
```

super()

super() gives you access to the methods of the superclass from the subclass.

super() alone returns a temporary object of the superclass that then allows you to call the superclass's methods.

Calling the previously built methods with super() saves you from writing those methods again in your subclass.

Eg: ①

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width

class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)

square = Square(4)
square.area()
```

→ output ⇒ 16

super() keyword example 2 :-

```
class Mammal:
    def __init__(self, mammal):
        print(mammal, "is a mammal")

class noFly(Mammal):
    def __init__(self, nofly):
        print(nofly, "cannot fly")
        super().__init__(nofly)

class noSwim(Mammal):
    def __init__(self, noswim):
        print(noswim, "cannot swim")
        super().__init__(noswim)

class Dog(noSwim, noFly):
    def __init__(self):
        print("Dog has 4 legs")
        super().__init__("Dog")

d = Dog()
```

output:

```
Dog has 4 legs
Dog cannot Swim
Dog cannot Fly
Dog is a mammal.
```

class method:

These are the methods that are bound to the class and not the objects.

They take class as their first instance parameter.

They have the access to change the state of the class at any instance.

static method :-

These are the methods that are bound to class and not the objects.

These methods can be called without an object for that class.

Which also means that they cannot modify the state of an object.

These are useful, when We don't have to pass the object a class as its parameter & where the function can be called only with whatever parameters are passed.

(like a utility)

```
class MyClass:
    def method(self):                    ——— Instance Method
        print("Instance Method")
    @classmethod
    def classmethod(cls):                ——— Class method
        print('————')
    @staticmethod
    def staticmethod():                  ——— Static method
```

Class Variable:

→ Declared inside the class definition, but outside of the instance methods

→ They are not tied to any particular object hence shared across all the obj of a class

→ Modifying a class variable affects all the objects instance at the same tim

Instance Variable

→ Declared inside the constructor method (--init

→ They are tied to particular object instance of the class.

→ Contents of an instance variable are completely independent from one object instance to other

Eg:- Class Car:

wheels = 4 ——————→ Class Variable

def __init__(self, name):

self.name = name ——→ Instance variable

Multithreading

A process is an instance of a computer program that is being executed.

A Thread is an entity within the process that can be scheduled for execution.
It is the smallest unit of processing in the

Thread is a sequence of instructions within the program that can be executed independently of other code.

- Each thread has;
  1) Thread ID
  2) Thread State
  3) Stack Pointer
  4) Parent process pointer
  5) Register set
  6) Program Counter.

Multithreading is the ability of the processor to execute multiple threads concurrently.
It allows you to break down an application into multiple sub-tasks and run these tasks simultaneously.

Multithreading is a technique in which multiple threads in a process share their data space with the main thread (program) which makes information sharing & the communication b/n the threads easy & efficient.

Thread functions:

~~isalive()~~
~~active_count()~~
~~current_thre~~

activeCount()
currentThread()
enumerate()
isAlive()
isDaemon()

start()
run()
join()

1) Deadlock :- Deadlock occurs when different threads try to access/acquire the shared resources at the same time. As a result, none of the processes get a chance to execute as they are waiting for another resource held by some other process

2) Race Condition

To deal with deadlock, race condition or any other time issues, threads need to be synchronized.

Ways to synchronize

1) Locks
2) RLocks
3) Semaphores
4) Conditions
5) Events
6) Barriers

Normal Thread Example

```
impost threading

def cube(l):
    print("cube:", l*l*l)

def square(l):
    print("square:", l*l)

if __name__=="__main__":
    t1 = threading.Thread(target=cube, args=(10,))
    t2 = threading.Thread(target=square, args=(10,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

---

join() :- Once the thread starts, the main program (main thread) also keeps on executing. In order to stop the execution of the main pgm until a thread is complete, we use join()

Running thread as a background daemon:

```
import threading
import os
import time
class DemoThread:
    def __init__(self, interval=1):
        self.interval = interval
        thread = threading.Thread(target=self.run,
                                  args=())
        thread.daemon = True
        thread.start()

    def run(self):
        while True:
            out = os.system("<commands>")
            print(out)
            time.sleep(interval)

d = DemoThread()
```

```
import threading
lock = threading.Lock()
def func1():
    for i in range(5):
        lock.acquire()
        print("Lock acquired")
        print("Func 1")
        lock.release()

def func2():
    for i in range(5):
        lock.acquire()
        print("Lock acquired")
        print("Func2")
        lock.release()

if __name__ == "__main__":
    t1 = threading.Thread(target=func1)
    t2 = threading.Thread(target=func2)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

Collections

Collection module in Python provides different type of containers.

Container is an object which has different type of data, which gives us to access and iterate over them.

1) Counter
2) Ordereddict
3) Chainmap
4) Namedtuple
5) deque

1) Counter :

Counter is a sub-class of dictionary.
It keeps count of each element in the iterable in the form of an unordered dictionary.

Key - represents the element
Value - represents the count of the element

Eg: ~~printe~~ from collections import Counter
print (Counter ([1, 2, 1, 3, 4, 1, 2]))

O/p:
Counter ({1:3, 2:2, 3:1, 4:1})

2) Ordereddict

It is a sub-class of dict, it remembers the order in which keys were added or removed.

Eg: from collections import Ordereddict

```
od = Ordereddict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
for k,v in od.items():
    print(k,v)
od.pop['a']
od['a'] = 10
for k,v in od.items():
    print(k,v)
```

O/P:

a 1
b 2
c 3
b 2
c 3
a 10

3) Chainmap → It encapsulates many dictionaries into single a single unit & returns ~~list of dictionary~~.

Eg:-
```
d1 = {'a':1, 'b':2}
d2 = {'c':3, 'd':4}
d3 = {'e':5}
cm = Chainmap(d1,d2)
print(cm)
cm2 = cm.make_child(d3)
print(cm2)
```

O/P: Chainmap({'a':1,'b':2},{'c':3,'d':4})
Chainmap({'a':1,'b':2},{'c':3,'d':4},{'e':5})

4) namedtuple

namedtuple returns a tuple object with names for each position.

Eg:-
```
from collections import namedtuple
student = namedtuple('Student', ['name', 'age', 'id'])
s1 = student('ABC', 18, 1)
print(s1.name)  ⟶ ABC
print(s1.age)   ⟶ 18
li = ['XYZ', 28, 2]
s2 = student._make(li)
print(s2.name)  ⟶ XYZ
print(s2.age)   ⟶ 28
```

5) deque : [Doubley - Ended - Queue]

It is optimized List for quicker append & pop operations from both sides; with time complexity O(1) [whereas for normal list - O(n)]
It takes list as an argument.

Eg:
```
from collections import deque
dq = deque([1,2,3,4])
print(dq)  ⟶ deque([1,2,3,4])
dq.append(5)
print(dq)  ⟶ deque([1,2,3,4,5])
dq.appendleft(10)
print(dq)  ⟶ [10,1,2,3,4,5]
dq.pop()
print(dq)  ⟶ [10,1,2,3,4]
dq.popleft()
print(dq)  ⟶ [1,2,3,4]
```

Abstract Class (ABC)

Data abstraction in Python is a programming concept that hides complex implementation details while exposing only essential information and functionalities to users.

```
from abc import ABC, abstractmethod

class Car(ABC):
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
    @abstractmethod
    def printDetails(self):
        pass

    def __(self)
        <  >

class Hatchback(Car):
    def printDetails(self):
        print("Brand:", self.brand)
        print("Model:", self.model)

car = Hatchback("Maruthi", "800")
car.printDetails()
```

Subprocess Module

subprocess module help in executing system command & return its output & error dat

1) import subprocess
output = subprocess.getoutput ("<command>")
print (output)

---

2) import subprocess
output = subprocess.check_output ("<command>", shell = True)
print (output)

---

3) import subprocess
process = subprocess.Popen (cmd, shell = True, stdout = subprocess.PIPE, stderr = supprocess.PIPE).communi
print ("Error: ", process[1].decode ("utf-8"))
list ⇐ print ("Output:", process[0].decode ("utf-8"))

---

1) shell = True ⇒ will run the given command on the shell Terminal

2) .communicate () ⇒ To interact with the proce
Send the data to stdin & close it.
Read the data from stdout & stderr, until end-of-file is reached. Wait for process to terminate.
*** Return a tuple (stdout, stderr)

json module

import json

json.load() → To read the json file

json.loads() → To parse a JSON string & return as a dictionary

json.dump() → To write the contents into a json file as json obj

json.dumps() → convert dict into json string

```
with open(filename) as json_file:
    data = json.load(json_file)
```

---

import yaml

```
with open(filename) as f:
    data = yaml.load(f, Loader=yaml.FullLoader)
```

load_all()

dump()

logging Module

```
import logging
def log_function(name):
    logging.basicConfig(filename="<filename>",
    format='%(filename)s - %(levelname)s -
        %(message)s', filemode='w')
    logger = logging.getLogger()
    logger.setLevel(logging.INFO)
    return logger
```